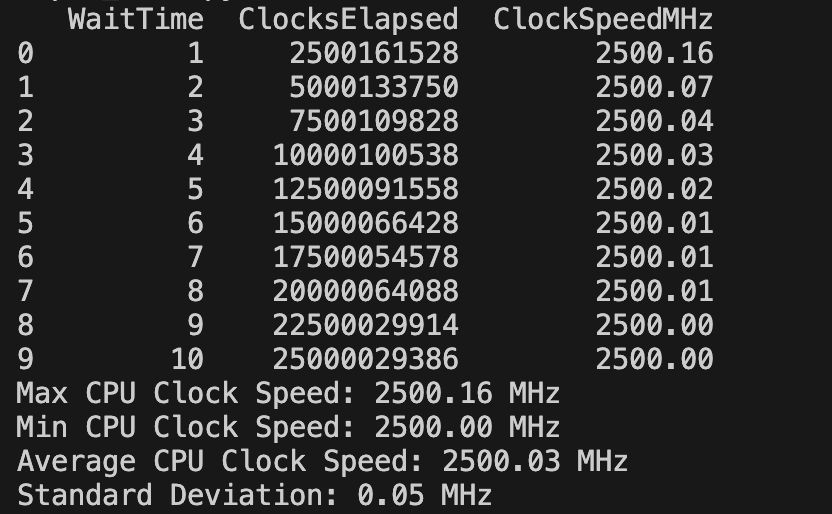
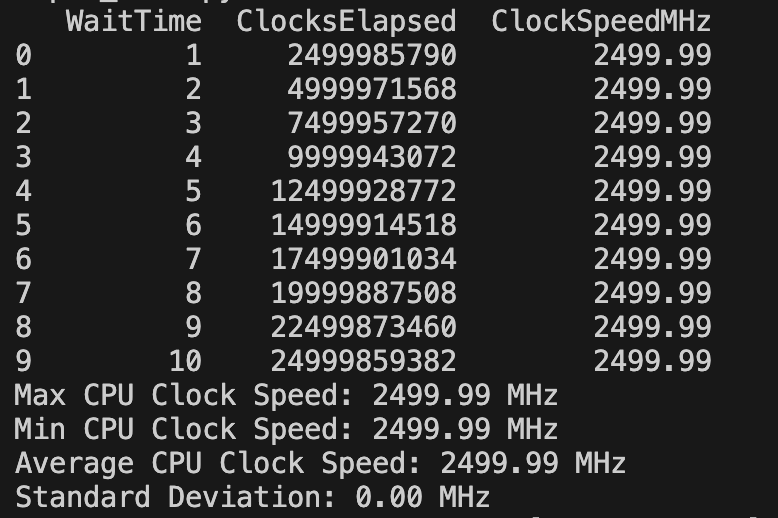
Problem 1 Eval

In this EVAL problem, you will evaluate the result of the clock measuring functions developed in the corresponding BUILD assignment. In particular, we will estimate which method (SLEEP vs. BUSYWAIT) works best for stable CPU speed estimates, and use the power of clock measurements to understand a bit how the SCC works under the hood.

**a)** First, estimate the CPU clock speed on your own machine using the SLEEP method. To do this, collect the results returned by the clock executable you implemented by running it 10 times. Start with a wait time of 1 second, all the way up to (and including) 10 seconds, increasing the wait time by 1 second at each step.  
From the 10 CPU clock speed measurements you have obtained, compute and report the max, min, average, and standard deviation.



**b)** Repeat the same procedure above where instead of the SLEEP method, you use the BUSYWAIT method for all the measurements. Then, compute and report the same statistics on the measured CPU clock speed.



**c)** Now let us this capability to answer the following question: is the SCC comprised of identical machines? In other words, does the SCC always allocate you a physical machine with the same type of CPUs? To answer this question, conduct 10 clock frequency measurements of 10 seconds each, but start each experiment in a different SCC session, with some time elapsing in between. For instance, conduct the first 10-seconds measurement in a SCC session started on Tuesday morning, the second in the afternoon, etc. Each time, make sure to start a new SCC session. In your report, produce a table with one column being the date/time of the experiment and resulting measured clock speed. Next, draw your conclusions about how the SCC works under the hood in terms of machine provisioning.

For this part of the evaluation, I aimed to test whether the Shared Computing Cluster (SCC) is comprised of identical machines or if different sessions may allocate different types of CPUs. Ideally, this would involve conducting the measurements at different times of the day across different sessions. However, due to time constraints, I modified the experiment slightly as follows (also seen on piazza):

1. I started by running up to 5 SCC sessions concurrently (the maximum limit). After completing the clock measurement in one session, I terminated it and started a new one to ensure each measurement was taken on a potentially different machine.
2. The clock frequency was measured using a busy-wait for 10 seconds in each of the 10 SCC sessions, with the results shown in the table above.

| Time | Server | Wait Time | Clocks Elapsed | Clock Speed MHZ |
| --- | --- | --- | --- | --- |
| 2024-09-18 16:40:42 EDT | 1 | 10 | 24999873926 | 2499.99 |
| 2024-09-18 18:01:48 EDT | 2 | 10 | 25999818951 | 2599.98 |
| 2024-09-18 18:02:02 EDT | 3 | 10 | ​​25999834215 | 2599.98 |
| 2024-09-18 18:02:13 EDT | 4 | 10 | 25999775088 | 2599.98 |
| 2024-09-18 18:02:23 EDT | 5 | 10 | 25999573242 | 2599.96 |
| 2024-09-18 18:11:26 EDT | 6 | 10 | 27934240118 | 2793.42 |
| 2024-09-18 18:11:57 EDT | 7 | 10 | 27934326778 | 2793.43 |
| 2024-09-18 18:14:26 EDT | 8 | 10 | 25939779339 | 2593.98 |
| 2024-09-18 18:16:56 EDT | 9 | 10 | 25939827077 | 2593.98 |
| 2024-09-18 18:19:24 EDT | 10 | 10 | 27934256944 | 2793.43 |

The measurements reveal variations in the clock speeds, ranging from approximately 2499.99 MHz to 2793.43 MHz. This indicates that the CPUs in use are not identical across different SCC sessions. Additionally, it seems that the servers that were launched close together in time (e.g., servers 2, 3, and 4) exhibit similar clock speeds, suggesting that they were likely allocated CPUs of the same type or from the same machine pool. In contrast, servers that were started with a longer time gap between their launch times (e.g., servers 5, 6, 7) showed variations in clock speeds, supporting the hypothesis that the SCC system may allocate different physical machines depending on availability and timing.

Based on the data, it is reasonable to conclude that the SCC does not always allocate the same type of physical CPU for each session. The observed variability in clock speeds indicates that the SCC likely uses a mix of CPUs within its infrastructure. Additionally, there appears to be a correlation between the time of session initiation and the type of CPU allocated. When multiple sessions are started around the same time, they are more likely to be allocated similar CPUs, potentially indicating a preference for using available resources from the same machine pool.

These findings suggest that the SCC's underlying provisioning system dynamically allocates available hardware based on current demand and resource availability, leading to potential differences in CPU performance characteristics between sessions.

Problem 2 Eval

In this EVAL problem, we will start to examine the behavior of the server process in response to variable traffic conditions generated by the client.  
HINT: many of the questions below, will ask you to determine a lapse of time in which the server was active. You can compute that time window in post-processing as the lapse of time between the <receipt timestamp> of the very first request and the <completion timestamp> of the last request processed by the server.

**a)** First, launch the client with the following parameters: client -a 10 -s 12 -n 500 <some port> where <some port> is the same port (of your choice) used for the server. Allow the client and server to exchange data and terminate, then post-process the output produced by the server.  
From the server output, compute the average throughput of the server in terms of requests per second. Walk us through the steps you performed to accomplish this.

**Steps Taken:**

**Throughput (requests per second) = total number of requests / Total active time window**

First, I ran these 2 terminal commands in side-by-side terminals:

* ./build/server 8080 > server\_output.log (saving to output\_log for easier parsing)
* ./client -a 10 -s 12 -n 500 8080

From the output log, we can extract the first and last packets sent.

**Format of each packet:**

R<request\_id>:<sent\_timestamp>,<request\_length>,<receipt\_timestamp>,<completion\_timestamp>.

**First Packet:**

R0:3233315.037737,0.041792,3233315.037783,3233315.079575

**Last Packet:**

R499:3233365.929910,0.063366,3233366.262623,3233366.325989

**Throughput (requests per second) = total number of requests / Total active time window**

**Total active time window** = 3233366.325989 - 3233315.037783 = 51.2882059999 seconds

**total number of requests** = 500

**Throughput (requests per second)** = 500 / 51.2882059999

**Throughput (requests per second)** = 9.74882997469 requests per second

**b)** Follow the same procedure above and once again post-process the output of the server. This time, compute the utilization of the server as a percentage. Walk us through the steps you performed to accomplish this.

**Steps Taken:**

**Server Utilization = (Total Busy Time / Total Active Time Window) \* 100.**

**As before:**

First, I ran these 2 terminal commands in side-by-side terminals:

* ./build/server 8080 > server\_output.log (saving to output\_log for easier parsing)
* ./client -a 10 -s 12 -n 500 8080

From the output log, we can extract the first and last packets sent.

**Calculate Total Active Time Window**:

* As in part (a), first we find the <receipt\_timestamp> of the first packet and the <completion\_timestamp> of the last packet to determine the total active time window:
  + Total Active Time Window = <completion\_timestamp> of the last packet - <receipt\_timestamp> of the first packet.

**Format of each packet:**

R<request\_id>:<sent\_timestamp>,<request\_length>,<receipt\_timestamp>,<completion\_timestamp>.

**First Packet:**

R0:3233315.037737,0.041792,3233315.037783,3233315.079575

**Last Packet:**

R499:3233365.929910,0.063366,3233366.262623,3233366.325989

**Total active time window** = 3233366.325989 - 3233315.037783 = 51.2882059999 seconds

**Calculate Total Busy Time**:

* The total time the server spent processing requests is the sum of the <request\_length> fields for all the requests.
* Here I plan to parse each log line to extract <request\_length> and sum these values using a Python script to get the total busy time.

**Script Used:**

| def parse\_server\_log\_for\_utilization(file\_path):  with open(file\_path, 'r') as file:  lines = file.readlines()   first\_receipt\_timestamp = None  last\_completion\_timestamp = None  total\_busy\_time = 0.0 # Accumulate busy time here   for line in lines:  # Split the line to extract timestamp information  parts = line.split(':')  if len(parts) < 2:  continue   # Further split to extract individual fields  timestamp\_info = parts[1].split(',')   # Extract the relevant fields  receipt\_timestamp = float(timestamp\_info[2])  completion\_timestamp = float(timestamp\_info[3])  request\_length = float(timestamp\_info[1]) # Request length in seconds   # Track the first and last timestamps  if first\_receipt\_timestamp is None:  first\_receipt\_timestamp = receipt\_timestamp  last\_completion\_timestamp = completion\_timestamp   # Sum the total busy time  total\_busy\_time += request\_length   # Calculate total active time window  total\_time = last\_completion\_timestamp - first\_receipt\_timestamp   # Calculate server utilization as a percentage  utilization = (total\_busy\_time / total\_time) \* 100   return utilization, total\_time, total\_busy\_time  # usage file\_path = 'server\_output.log' utilization, total\_time, total\_busy\_time = parse\_server\_log\_for\_utilization(file\_path)  print(f"Total Time Window: {total\_time} seconds") print(f"Total Busy Time: {total\_busy\_time} seconds") print(f"Server Utilization: {utilization}%") |
| --- |

**Output:**



**c)** Repeat the experiment above but by running the server and client back-to-back, while at the same time extracting some statistics about the server run from the OS. To do so, move to the build directory and use the following command line:

| /usr/bin/time -v ./server 2222 & ./client -a 10 -s 12 -n 500 2222 |
| --- |

NOTE: by doing so, BOTH client and server will run in the same terminal and their output will be interleaved on the console. If you want to suppress the output of the client, redirect the stdout of the client to the null device using:

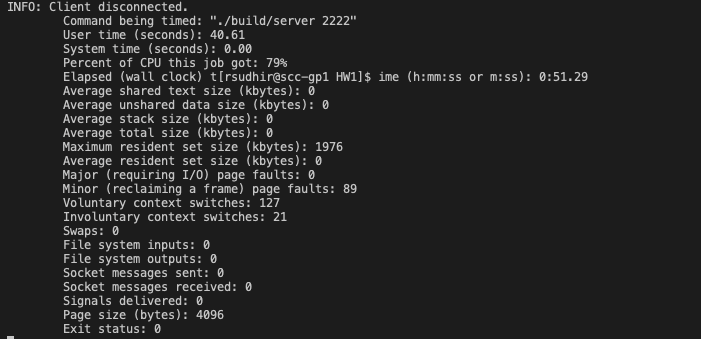
| /usr/bin/time -v ./server 2222 & ./client -a 10 -s 12 -n 500 2222 > /dev/null |
| --- |

Also note that the server will run in background. You can use the following command to kill it if it does not terminate:

| killall server |
| --- |

After the server has run successfully, the time utility will print a host of statistics acquired from the OS. Take a look at the entry “Percent of CPU this job got:”. Does it match with what you computed in the previous question?

**Statistics Printed**

****

Yes, it does seem to line up neatly with what i received in part b.

**d)** Now let us repeat the computation of the utilization of the server as in Qb) but this time sweep through the -a parameter passed to the client. In particular, run the first experiment for a value of 1; then a second time with a value of 2; and so on until and including the case where the value is 12. Thus, you will run 12 experiments in total.  
   
Produce a plot that depicts the trend of the resulting server utilization (y-axis) as a function of the arrival rate (-a) parameter (x-axis). Is there any correlation between the two values?

**Steps taken:**

First ill run the client and server 12 times, varying the -a parameter of the client from 1 to 12

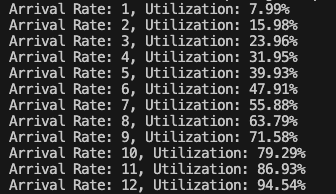
| ./build/server 2222 > server\_output\_<arrival\_rate>.log & ./client -a <arrival\_rate> -s 12 12 -n 500 2222 > /dev/null |
| --- |

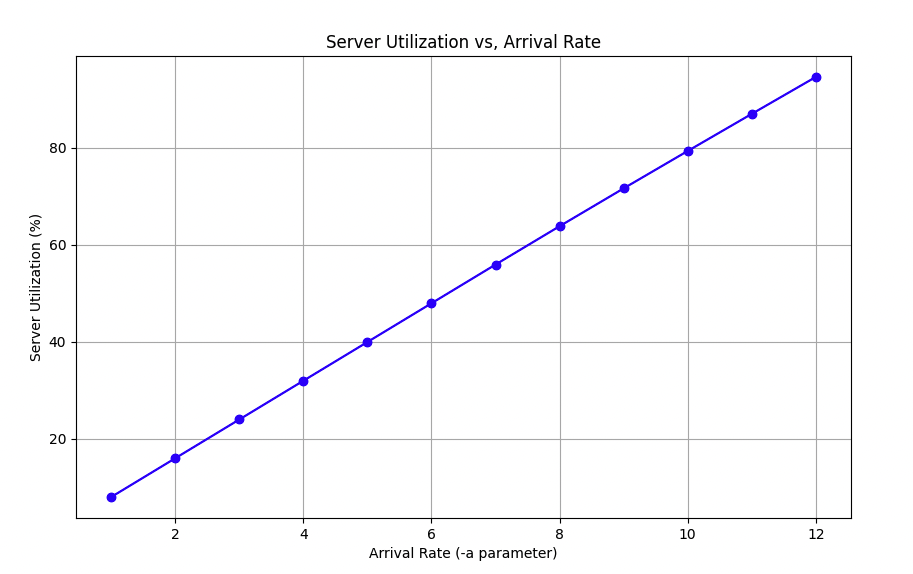
After each run ill extract the server utilization using the same approach i used in part b/c, although i aim to modify the script a little to make this easier. What I did was run the command 12 times, saving each output in a folder. This allowed me to loop through the data and calculate utilization for each arrival rate, i could then plot this.

**Script Used to extract data:**

| import matplotlib.pyplot as plt  def parse\_server\_log\_for\_utlization(file\_path):  with open(file\_path, 'r') as file:  lines = file.readlines()   first\_receipt\_timestamp = None  last\_completion\_timestamp = None  total\_busy\_time = 0.0 # accumulates busy time   for line in lines:   # split the line to extract time stamp info   parts = line.split(':')  if len(parts) < 2:  continue   # Further split to extract individual fields  timestamp\_info = parts[1].split(',')   # Check if the line contains the expected number of fields  if len(timestamp\_info) < 4:  continue    try:  # Extract the relevant fields  receipt\_timestamp = float(timestamp\_info[2])  completion\_timestamp = float(timestamp\_info[3])  request\_length = float(timestamp\_info[1]) # Request length in seconds   # Track the first and last timestamps  if first\_receipt\_timestamp is None:  first\_receipt\_timestamp = receipt\_timestamp  last\_completion\_timestamp = completion\_timestamp   total\_busy\_time += request\_length   except ValueError:  continue    # Checking if we valid timestamps to process  if first\_receipt\_timestamp is None or last\_completion\_timestamp is None:  return None, None, None    # Calculate total active time window  total\_time = last\_completion\_timestamp - first\_receipt\_timestamp   # Calculate server utilization as a percentage  utilization = (total\_busy\_time/total\_time) \* 100   return utilization, total\_time, total\_busy\_time   arrival\_rates = range(1, 13) # 1 to 12 utilizations = []  for arrival\_rate in arrival\_rates:   file\_path = f'./eval\_server\_output/server\_output\_{arrival\_rate}.log' # for each of the server outputs   utilization, total\_time, total\_busy\_time = parse\_server\_log\_for\_utlization(file\_path)   if utilization is not None:  utilizations.append(utilization)  print(f"Arrival Rate: {arrival\_rate}, Utilization: {utilization:.2f}%")  else:  utilizations.append(0)  print(f"Arrival Rate: {arrival\_rate}, Utilization: Could not be calculated.")  # plot the results plt.figure(figsize=(10,6)) plt.plot(arrival\_rates, utilizations, marker='o', linestyle='-', color='b') plt.title('Server Utilization vs, Arrival Rate') plt.xlabel('Arrival Rate (-a parameter)') plt.ylabel('Server Utilization (%)') plt.grid(True) plt.show() |
| --- |

**Output:**





From the plot, we can clearly see a strong positive correlation between the arrival rate and server utilization. This probably plateaus after a certain amount, i.e., when the server utilization becomes 100%.

**e)** By reusing the same exact output files produced as part of the previous question, let us reason about the response time of the various requests. First, compute the average response time for all the 500 requests as observed in the case when the parameter -a was set to 10. Also compute max, min, and std. deviation.

To compute the average response time, along with the max, min, and standard deviation for the 500 requests when the -a parameter is set to 10, I will need to extract the response times from the server output logs. Here's the approach I will follow:

Extract Response Times:

* The response time for each request is the difference between <completion\_timestamp> and <sent\_timestamp> for that request.
* I will calculate the response time for all 500 requests when -a is set to 10 (from server\_output\_10.log).

Compute Statistics:

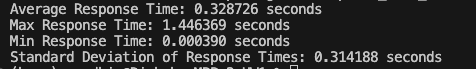
* Average: The mean of all response times.
* Max: The largest response time.
* Min: The smallest response time.
* Standard Deviation: The measure of how much the response times vary.

**Again I can use a script here to accomplish this:**

**Script Used:**

| import numpy as np  def parse\_response\_times(file\_path):  # list to store response times  response\_times = []   with open(file\_path, 'r') as file:  lines = file.readlines()   for line in lines:  # Split the line to extract timestamp information  parts = line.split(':')  if len(parts) < 2:  continue   # Further split to extract individual fields  timestamp\_info = parts[1].split(',')   # Check if the line contains the expected number of fields  if len(timestamp\_info) < 4:  continue   try:  # Extract the relevant fields  sent\_timestamp = float(timestamp\_info[0]) # Sent\_timestamp  completion\_timestamp = float(timestamp\_info[3]) # Completion timestamp   # Calculate response time  response\_time = completion\_timestamp - sent\_timestamp  # Append response time  response\_times.append(response\_time)   except ValueError:  continue   return response\_times   # Path to the log file where -a was set to 10 file\_path = './eval\_server\_output/server\_output\_10.log'  # Get response times response\_times = parse\_response\_times(file\_path)  # Compute statistics if response times are available if response\_times:  avg\_response\_time = np.mean(response\_times)  max\_response\_time = np.max(response\_times)  min\_response\_time = np.min(response\_times)  std\_dev\_response\_time = np.std(response\_times)   # Print the results  print(f"Average Response Time: {avg\_response\_time:.6f} seconds")  print(f"Max Response Time: {max\_response\_time:.6f} seconds")  print(f"Min Response Time: {min\_response\_time:.6f} seconds")  print(f"Standard Deviation of Response Times: {std\_dev\_response\_time:.6f} seconds") else:  print("No response times found in the log file.") |
| --- |

**Output:**

****

**f)** Now, repeat the average response time calculation individually for the various runs with the -a parameter from 1 to 12. Finally, a plot will be produced that depicts the trend of the average response time y-axis as a function of the server utilization x-axis. What relationship do you discover between the two quantities? Are they directly or inversely proportional to each other? Is the relationship linear?

To complete this task, Ill need to calculate the average response time for each of the runs with the -a parameter ranging from 1 to 12 and then plot the average response time on the y-axis as a function of the server utilization on the x-axis. Finally, we will analyze the relationship between response time and utilization.

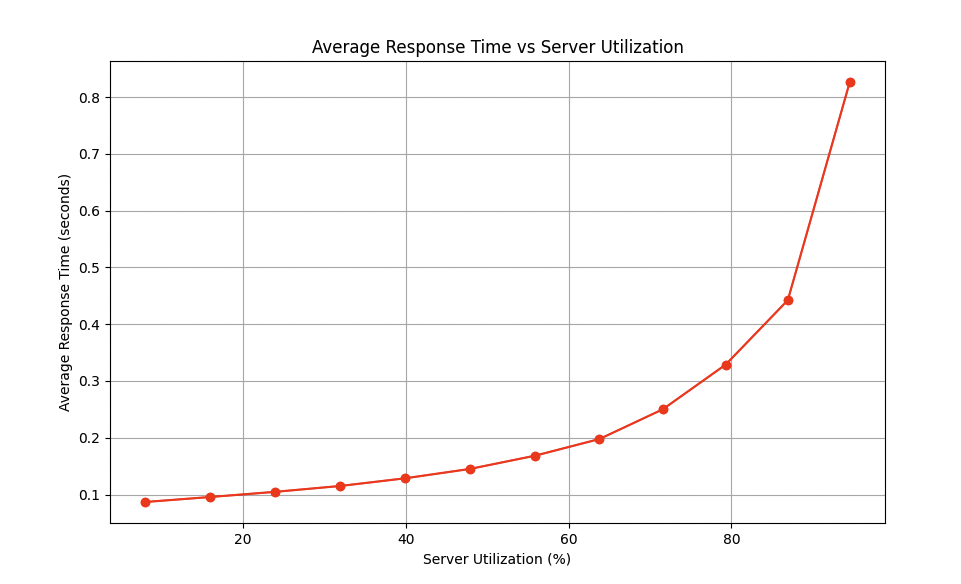
1. **Calculate Average Response Time for Each -a Value**:
   * Use the same method as in Part (e) to compute the **average response time** for each run where -a is set from 1 to 12.
2. **Reuse Server Utilization Data**:
   * Reuse the **server utilization** values I calculated in Part (d) for each -a value.
3. **Plot the Results**:
   * Plot the **average response time** (y-axis) against the **server utilization** (x-axis).

**Script Used:**

| import numpy as np import matplotlib.pyplot as plt  def parse\_response\_times(file\_path):  response\_times = []   with open(file\_path, 'r') as file:  lines = file.readlines()   for line in lines:  parts = line.split(':')  if len(parts) < 2:  continue   timestamp\_info = parts[1].split(',')  if len(timestamp\_info) < 4:  continue   try:  sent\_timestamp = float(timestamp\_info[0])  completion\_timestamp = float(timestamp\_info[3])   response\_time = completion\_timestamp - sent\_timestamp  response\_times.append(response\_time)   except ValueError:  continue   return response\_times   # Arrival rates from 1 to 12 arrival\_rates = range(1, 13) avg\_response\_times = [] server\_utilizations = [7.99, 15.98, 23.96, 31.95, 39.93, 47.91, 55.88, 63.79, 71.58, 79.29, 86.93, 94.54] # utilization values from Part d  # Process each run (for each `-a` value from 1 to 12) for arrival\_rate in arrival\_rates:  file\_path = f'./eval\_server\_output/server\_output\_{arrival\_rate}.log'   response\_times = parse\_response\_times(file\_path)   if response\_times:  avg\_response\_time = np.mean(response\_times)  avg\_response\_times.append(avg\_response\_time)  print(f"Arrival Rate: {arrival\_rate}, Avg. Response Time: {avg\_response\_time:.6f} seconds")  else:  avg\_response\_times.append(0)  print(f"Arrival Rate: {arrival\_rate}, Avg. Response Time: Could not be calculated.")  # Plotting the average response time vs server utilization plt.figure(figsize=(10, 6)) plt.plot(server\_utilizations, avg\_response\_times, marker='o', linestyle='-', color='r') plt.title('Average Response Time vs Server Utilization') plt.xlabel('Server Utilization (%)') plt.ylabel('Average Response Time (seconds)') plt.grid(True) plt.show() |
| --- |

**Output:**

****

****

From the experiment, we observe a clear relationship between **server utilization** and **average response time** as the client arrival rate increases. Specifically:

* **Increasing Arrival Rate**: As the client arrival rate (-a parameter) increases from 1 to 12, the number of requests sent to the server increases, leading to higher demand on the server's resources.
* **Increasing Server Utilization**: As the arrival rate increases, the server becomes increasingly utilized. This makes sense, as more incoming requests put greater pressure on the server to process them in a timely manner.
* **Increasing Average Response Time**: The average response time also increases as server utilization grows. When the server is handling fewer requests (low utilization), the response time is short because the server has sufficient resources to process each request promptly. However, as the server becomes more heavily loaded (high utilization), it takes longer to process each request due to resource contention, leading to increased response times.

It seems the graph has an exponential relationship till max utilization.